Formal semantics, modular specification, and symbolic verification of product-line behaviour

Classen, Andreas; Cordy, Maxime; Heymans, Patrick; Legay, Axel; Schobbens, Pierre Yves

Published in:
Science of Computer Programming

DOI:
10.1016/j.scico.2013.09.019

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Science of Computer Programming 80 (2014) 416–439

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formal semantics, modular specification, and symbolic verification of product-line behaviour ☆

Andreas Classen [a], Maxime Cordy [a,*,1], Patrick Heymans [a,b], Axel Legay [c], Pierre-Yves Schobbens [a]

[a] PReCISE Research Center, University of Namur, Belgium
[b] INRIA Lille-Nord Europe, Université Lille 1 – LIFL – CNRS, France
[c] INRIA Rennes, France

HIGHLIGHTS

- We use Featured Transition Systems (FTS) to model Software Product Lines (SPLs).
- We design *symbolic* algorithms for checking an FTS against temporal properties.
- We give a new compositional formal semantics to the fSMV language.
- We prove the expressiveness equivalence between fSMV and FTS.
- We evaluate practical implications of our results through our toolset and case study.

ARTICLE INFO

Article history:
Received 5 May 2012
Received in revised form 22 August 2013
Accepted 18 September 2013
Available online 22 October 2013

Keywords:
Software product line
Verification
Feature
Language
Specification

ABSTRACT

Formal techniques for specifying and verifying Software Product Lines (SPL) are actively studied. While the foundations of this domain recently made significant progress with the introduction of Featured Transition Systems (FTSs) and associated algorithms, SPL model checking still faces the well-known state explosion problem. Moreover, there is a need for high-level specification languages usable in industry. We address the state explosion problem by applying the principles of symbolic model checking to FTS-based verification of SPLs. In order to specify properties on specific products only, we extend the temporal logic CTL with feature quantifiers. Next, we show how SPL behaviour can be specified with fSMV, a variant of SMV, the specification language of the industry-strength model checker NuSMV. fSMV is a feature-oriented extension of SMV originally introduced by Plath and Ryan. We prove that fSMV and FTSs are expressively equivalent. Finally, we connect these results to a NuSMV extension we developed for verifying SPLs against CTL properties.

1. Introduction

Software Product Lines (SPLs) are a popular software engineering paradigm that seeks to maximise reuse by planning upfront which *features* should be common, resp. variable, for several similar software systems [17]. The different systems in an SPL (called "products") are identified in advance and a model of their differences and commonalities is created. This model is usually a *feature diagram* [29,39], features being atomic units of difference that appear natural to stakeholders and

☆ This article is an extended version of the paper *A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, Symbolic model checking of software product lines, in: Proceedings of ICSE '11, ACM, 2011, pp. 321–330.*

* Corresponding author.
E-mail addresses: acs@info.fundp.ac.be (A. Classen), mcr@info.fundp.ac.be (M. Cordy), phe@info.fundp.ac.be (P. Heymans), axel.legay@inria.fr (A. Legay), pys@info.fundp.ac.be (P.-Y. Schobbens).

[1] FNRS research fellow, project FC 91490.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.09.019

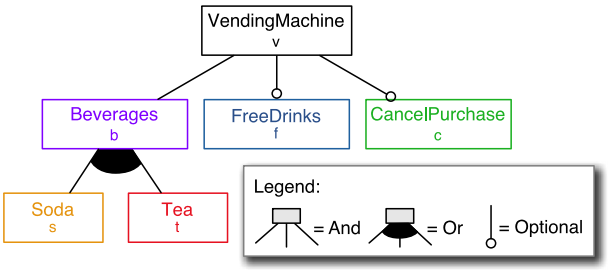

Fig. 1. The feature diagram of a vending machine.

technicians alike [13]. A toy example of a feature diagram is given in Fig. 1. It models a vending machine SPL with five features. Feature *Beverages* is mandatory, while *FreeDrinks* and *CancelPurchase* are optional. *Soda* and *Tea* are subfeatures of *Beverages*, and any product must have at least one of them.

In the real world, SPL development is increasingly applied to embedded and critical systems [21]. Formal modelling and verification of SPL behaviour are thus vital for quality assurance and are actively studied [24,31,32,15]. Model checking is a well-known automatic technique for verifying both hardware and software. It allows to verify desired behavioural properties on a model of a given system [4]. For example, an intended property for the vending machine is: "A customer can always cancel a purchase before the beverage is served". In the context of single systems, a model checker returns true if a property is satisfied, or a counter-example (i.e. an execution trace) if it is violated. The model checking problem in SPL is different from the one in single systems engineering: an algorithm has to check all products against a property and pinpoint those products that violate it [16]. In our example, every vending machine without the *Cancel* feature would not satisfy the aforementioned property. The model checking problem is also harder, as it has to deal with the fact that there can be exponentially many, $O(2^{\#features})$, products to verify. In [15,9], we addressed the model checking problem for SPLs by introducing *Featured Transition Systems* (*FTSs*), a formalism to express the behaviour of all products of the SPL in one model. FTSs are transition systems [4] in which transitions are labelled with features (in addition to being labelled with actions). This allows one to keep track of the different products. We also proposed new model checking algorithms [15] that exploit the structure of the FTS and try to avoid an exponential number of verifications by exploring the FTS rather than the transition system of each individual product. Those algorithms can be used to verify properties expressed in Linear Time Logic (LTL). We call them *FTS algorithms*. The experimental results gathered so far show that this new approach is more efficient than an enumerative method that verifies each product individually. More information can be found on our project website http://www.info.fundp.ac.be/fts.

The main drawback of our previous FTS algorithms is that they rely on an explicit enumeration of the state space. Albeit we already observed that FTSs drastically reduce the time needed to verify the products of an SPL, they may still suffer from the state explosion problem. Overcoming this issue is a well-known challenge in explicit state space model checking. Symbolic algorithms, which make use of symbolic representations of the state space, are a solution to this problem. They have shown to be particularly efficient in the context of single-systems model checking and made possible the verification of huge systems [34].

Moreover, FTSs is a foundational formalism, not meant to be used directly by engineers. It is thus important to relate the FTS language to high-level languages that can be used in industrial settings. A suitable language for SPLs is *fSMV*, which was introduced by Plath and Ryan [36]. fSMV extends SMV (i.e. the (Nu)SMV model checker's input language) with primitives that allow to account for the addition of new features. More precisely, in fSMV, an SPL is represented by a base system described in the SMV language. Each additional feature is described independently, stating its assumptions and modifications. A product is built from the base system by adding features in a certain order.

In [36], Plath and Ryan propose a procedure to verify properties of a product. The verification procedure exploits the fact that a product can be expressed with SMV alone, and hence semantically as a *transition system*. This property allows them to reuse the classical symbolic verification procedure implemented in the (Nu)SMV toolset, which provides an efficient way to verify *a single product*. An fSMV model represents several products, each being the combination of the base system and a set of features. However, with the approach described in [36], one check per product is needed, which decreases the performance of the approach considerably. This is because transition systems do not allow to distinguish between features and hence between products. There is thus a need for a translation from fSMV to a formal model that is more suited to represent SPLs.

The contribution of the present paper is twofold. First, we propose symbolic algorithms for model checking an FTS against temporal properties. Second, we study the relation between FTSs and fSMV and provide them with a bi-directional translation. For this purpose, we provide the fSMV language with a definition different from the one given by Plath and Ryan [36]. The formal definition of fSMV is different from the one of SMV as we now have to characterise features and feature composition. Then, we show that FTSs and fSMV are equally expressive, that is, any FTS can be translated into an equivalent (in terms of behaviour) fSMV model and vice versa. This proof provides evidence that fSMV is an appropriate notation to model SPL behaviour. It is also a reference against which implementations can be proven correct, which is necessary to obtain trustworthy verification tools. Finally, we exploit this result to extend the NuSMV model checker with

FTS-based verification algorithms and we evaluate their efficiency empirically. We thus contribute to increasing the practical applicability of the FTS approach by yielding the first complete chain from high-level specification to efficient verification of SPL behaviour.

1.1. Structure of the paper

Section 2 introduces FTSs. Section 3 presents the feature Computation Tree Logic and discusses symbolic algorithms for verifying an FTS against a property expressed in this logic. Section 4 proposes a formalisation of the fSMV language. In this section, we also show that fSMV and FTSs are equally expressive. In Section 5, we describe how these results are exploited in our implementation, i.e. an extension of the NuSMV model-checker. In Section 6 we use an elevator system specification to illustrate and evaluate the benefits of the approach. Section 7 compares our work with other existing approaches. Section 8 wraps up the paper and discusses some perspectives for future research.

The paper is an extended version of [14]. In addition to added detail throughout, the main differences are in the vastly expanded Sections 4 and 5. In Section 4, we provide a formal description of the semantics of fSMV, as well as theoretical results relating fSMV and FTSs concluding in a proof that fSMV and FTSs are equivalent. In Section 5, we provide a detailed description of the implementation, including usage scenarios and examples.

2. Background

This section recapitulates the basic concepts and definitions that will be used through the rest of the paper. We start with a brief introduction to behavioural modelling of SPLs and FTSs in particular. We refer to [15,14,12] for additional information about FTSs.

Features are commonly used as atomic units of difference between the products that constitute an SPL. Let F be the set of features. A *product* p is specified by a set of features $p \subseteq F$. An SPL is a set of products, hence a set of sets of features $px \subseteq \mathcal{P}(F)$. Observe that not necessarily all combinations of features are *valid*. Indeed, some features might be incompatible, some features may depend on others, and so on. To represent the constraints on the set of products in a concise and structured way, the SPL community commonly uses *feature diagrams* (FD) (see, e.g. [29,39]). However, for the purpose of this paper, we do not need to restrict ourselves to feature diagrams and just assume a *feature model* defined as follows.

Definition 1. A feature model d is a tuple (N, px), where $N \subseteq F$ is the set of features and $px \subseteq \mathcal{P}(N)$ is the set of products. We also write $[\![d]\!]$ to denote px.

Given that we model a product as a set of features, we can encode a product with a *feature expression*, i.e. a Boolean function over the set of features. Any set of products can then be encoded as the feature expression resulting from the disjunction of the feature expressions encoding its products. Using feature expressions, we are thus able to encode the valid products of a feature model as a Boolean formula.

Usually, an SPL consists of millions of products that differ by a couple of features only. A major challenge, especially when looking for a verification procedure, is thus to find a compact representation for behaviour, that takes advantage of these repetitions. In [15], we have proposed FTSs, a concise formalism for representing SPL behaviours.

The behaviour of a single product can be represented by a *Transition System* (TS) [4]. A TS is a directed graph whose transitions are labelled with actions, and whose states are labelled with atomic propositions.

Definition 2. A TS is a tuple $ts = (S, Act, trans, I, AP, L)$, where

- S is a set of states and Act is a set of actions,
- $trans \subseteq S \times Act \times S$ is a set of transitions, with $(s_j, \alpha, s_k) \in trans$ sometimes noted $s_j \xrightarrow{\alpha} s_k$,
- $I \subseteq S$ is the set of initial states, AP is a set of atomic propositions, and $L : S \to \mathcal{P}(AP)$ is a labelling function.

In this paper, we assume that a TS has no terminal state, i.e. a state without outgoing transition. The semantics of a TS is a set of paths, i.e. sequences of states $s_0 = i, s_1, s_2, \ldots$ such that $i \in I$ and for each $j \geqslant 1$, there exists $\alpha \in Act \bullet (s_{j-1}, \alpha, s_j) \in trans$.

Note that the semantics of a TS is defined regardless of actions. If we use actions on transitions in our examples, it is just to make their meaning more apparent.

Let us now introduce FTSs [15]. An FTS is basically a TS with an additional function that labels transitions with Boolean functions over the features, called *feature expressions*. Those functions are used to introduce variability.[2] Formally:

[2] The definition of FTSs we give here is a slight generalisation over [15], where feature expressions were encoded by feature labels and priorities. This new definition has first been proposed in [14].

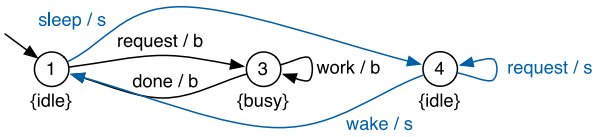

Fig. 2. Example FTS of a server.

Definition 3. An FTS is a tuple $fts = (S, Act, trans, I, AP, L, d, \gamma)$, where

- $S, Act, trans, I, AP, L$ are defined as in Definition 2,
- $d = (N, px)$ is a feature model as defined in Definition 1,
- $\gamma : trans \to (\{\bot, \top\}^{|N|} \to \{\bot, \top\})$ is a total function, labelling each transition with a feature expression. By $[\![\gamma(t)]\!]$, we denote the set of products that satisfy $\gamma(t)$, that is, those that contain transition t.

Note that $\top$ represents the Boolean constant *true* and $\bot$ the constant *false*, which means that $\{\bot, \top\}^{|N|} \to \{\bot, \top\}$ is the set of all functions over N Boolean parameters, one for each feature. In this paper, such functions are generally represented by *feature expressions*.

An FTS is a compact representation for a set of TS (that is, one TS for each product). For each transition, the feature expression determines for which products it exists. The TS corresponding to a product can be obtained by removing each transition whose feature expression is not satisfied by the product. This operation is called *projection*.

Definition 4 *(Projection in FTSs).* The projection of an FTS *fts* to a product $p \in [\![d]\!]$, noted $fts_{|p}$, is the TS $t = (S, Act, trans', I, AP, L)$ where $trans' = \{t \in trans \mid p \in [\![\gamma(t)]\!]\}$.

In this paper, we assume that any FTS is such that the projection onto any valid product has no terminal state. As we shall see later, the FTSs constructed in fSMV satisfy this assumption by construction.

A simple example of FTSs is given in Fig. 2. It depicts an SPL of controllers with two features. We consider that *basic* (abbreviated *b*) is a mandatory feature and covers the normal behaviour, and *sleep* (abbreviated *s*) is an optional feature. The feature model following Definition 1 would thus have the feature set $\{b, s\}$ and the products $\{b\}$ and $\{b, s\}$. The state space of the FTS is $\{1, 2, 3\}$. Each state is labelled with atomic propositions, namely a subset of $\{idle, busy\}$. Each transition is labelled with an action and a feature expression, separated by a "/". For example, we have $(1, sleep, 3) \in trans$ and $\gamma(1, sleep, 3) = s$. The feature expression of this transition is s, which means that it is only there for products that contain the feature s (the abbreviated name of the *sleep* feature). Since feature s was defined to be optional in the feature diagram, there is indeed a product, $\{b\}$, which does not contain this transition. Note that the information conveyed by the feature expressions is also represented by the use of a different colour for each feature.

Intuitively, the behaviour modelled in this example is the following. The controller starts in an idle state where it accepts requests. Once a request is received, the controller moves to the busy state. After completing the request, it returns to the idle state. In presence of the s feature, the controller can additionally enter state 3 where requests are ignored. This FTS represents the TSs of two products: $\{b\}$ and $\{b, s\}$.

The semantics of an FTS is given as a function that associates any valid product with its behaviour within the FTS [19]. It is obtained by first projecting the FTS to each product, resulting in a set of TSs, and then associating every product with the semantics of its corresponding TS. This implies that, as for TS, the semantics of an FTS does not consider actions. A more detailed introduction to FTSs can be found in [15,14,12]. A collection of examples that illustrate the concept is presented in [9].

3. Symbolic model checking of FTS

The algorithms presented in our previous paper [15] enumerate and visit system states *one by one*. Their aim is to mitigate the additional complexity that is due to the use of features in FTSs. They still face the state explosion problem as they do visit all states of the system one by one. An existing solution to this problem in single system model checking is symbolic model checking, that is, the use of symbolic representations of the state space. In this section, we combine FTSs and symbolic model checking to tackle both the aforementioned sources of complexity at once.

3.1. The feature Computation Tree Logic

Classical logics such as the Computation Tree Logic [7] can readily be used to express properties of products of an SPL. In particular, CTL expresses properties about a tree of executions. This tree is obtained when the common prefixes of all executions of a transition system are collapsed. It branches every time there is a choice between two transitions. In CTL, temporal operators are preceded by a path quantifier. The two quantifiers are E, requiring at least one path to satisfy a property; and A, requiring all paths to satisfy a property. In addition to the usual Boolean connectives, CTL has temporal

operators. Those are *next*, $X\ \phi$, which requires that the next state satisfies ϕ; *until*, $\phi_1 U \phi_2$, which requires that ϕ_2 is satisfied in some future state and that ϕ_1 holds until then; and *always*, $G\ \phi$, which requires that any future state satisfies ϕ.

Definition 5. A CTL formula ϕ is an expression of the form

$$\phi ::= \top \mid a\ (\in AP) \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid EX\ \phi \mid E(\phi_1 U \phi_2) \mid EG\ \phi.$$

This definition covers CTL properties in existential normal form. The A path quantifier can be obtained from E, and will thus not be explicitly considered in our algorithms. Moreover, you can derive conjunction as usual, as well as the *finally* operator, $EF\ \phi \triangleq E(\top U \phi)$, which requires that ϕ is satisfied in some future state.

An SPL is a set of systems which are not fundamentally different from systems developed as single systems from the outset. The properties that one would like individual products to satisfy (deadlock freedom, respect of a critical section, request/answer patterns, and so on) are the same as those that have been subject to model checking in single systems. We view model checking as orthogonal to other activities in SPLE. If model checking an SPL against a property corresponds to model checking all its products against this property, it would seem quite natural that the properties of interest to us are those that are used for single systems. And just as in single systems, we expect properties to be expressed with temporal logic formulae like CTL.

However, as a property might not be relevant to all products, a means to express the products for which a property holds should be added to temporal logics. This *quantifier* does not affect the semantics of the temporal property, but rather limits the range of products over which it holds. As an example, consider the property for the vending machine example: "*After the customer selects tea, the machine will eventually serve tea.*" The property is independent of all but the *Tea* feature, as products without this feature do not allow customers to select tea. That is, the property is irrelevant for products without the *Tea* feature; it has to hold only for those with the feature.

To specify the quantifier, we chose to use feature expressions. They are already prevalent in all of FTSs and fully expressive w.r.t. the set of products. This leads us to define *feature CTL* (fCTL) as follows.

Definition 6. An fCTL formula ϕ is an expression $\phi := [\chi]\phi' \mid \psi_1 \wedge \psi_2$ where ϕ' is a CTL formula, χ is a feature expression, ψ_1 and ψ_2 are fCTL formulae.

In fCTL, the example formula can be expressed as follows:

$$[\neg \mathit{FreeDrinks}]\ AG\ (\mathit{selected} \Rightarrow AF\ \mathit{open}).$$

Since we want to identify all the valid products that do not satisfy a given property, we are led to a new definition of the satisfiability relation, which is not Boolean anymore [18].

Definition 7. The F-satisfiability of the fCTL formula $[\chi]\Phi$ by an FTS with feature model d, noted $fts \models_F [\chi]\Phi$, is defined as the set

$$\left(fts \models_F [\chi]\Phi\right) = \left\{p \in [\![d]\!]_{FD} : p \in [\![\chi]\!] \Rightarrow fts_{|p} \models \Phi\right\}$$

that is, the set of products that yield a transition system satisfying the CTL formula Φ.

A CTL formula (without quantifier) is interpreted over an FTS as the fCTL formula quantified over all the products.

Definition 8. For a CTL formula ϕ, $fts \models \phi \triangleq fts \models [\top]\phi$.

Note that quantifiers in this logic can occur only at top-level. In this regard, our definition of fCTL differs from the one we gave in [14] where nesting of quantifiers was allowed. However, in actual examples, we found no need for the nesting of such quantifiers, which is why we decided to limit our logics to top-level quantifiers. This change does not affect or simplify the model checking algorithms, as we shall see later.

3.2. Symbolic model checking of fCTL properties

Symbolic model checking algorithms are based on fixed-point computations, making them rather different from the explicit search approach of our previous paper [15]. In the symbolic setting, sets of states and the transition relation are encoded directly with their characteristic functions. As we already said, characteristic functions can be represented by Binary Decision Diagrams (BDDs). We proceed in two steps. First, we describe a symbolic encoding for FTSs, before we proceed to the algorithms.

3.2.1. Encoding FTSs symbolically

We assume the existence of a binary encoding of states, that is, a function $enc : S \to \{\bot, \top\}^k$, where k is chosen large enough to encode all states. Given a product p, we also use the notation $enc(p)$ to denote the encoded product. With this encoding, $\{\bot, \top\}^k$ implicitly denotes the sets of all (encoded) states and $\{\bot, \top\}^n$, where n is the number of features, the set of all (encoded) products. When defining characteristic functions, we use the subscript notation, i.e., '$X \cup Y$' in $\chi_{X \cup Y}$, to denote the set for which this is the characteristic function. In parentheses follow the variables on which the function is defined. By convention, $\bar{s}$ (resp. $\bar{p}$) denotes a vector of Boolean variables encoding a state (resp. product). The *cofactor* $\chi_{T[\bar{s} \leftarrow enc(x)]}$ of a Boolean function $\chi_T(\bar{s}, \bar{p}, \ldots)$ is the Boolean function over the variables $\bar{p}, \ldots$ obtained by replacing the variables $\bar{s}$ by the value they take in $enc(x)$.

With the notation in place, we now show how an FTS can be encoded symbolically. The set of states is represented by a Boolean function χ_S and the set of initial states by χ_I. As usual, the labelling of states with atomic propositions L is represented by recording for each atomic proposition $a \in AP$ the set of states χ_a that are labelled by the proposition:

$$\chi_a(\bar{s}) : \{\bot, \top\}^k \to \{\bot, \top\} \bullet \chi_a\big(enc(s)\big) = \top \iff a \in L(s).$$

The transition relation is represented by a function that takes two encoded states (start and end) and an encoded product, and returns $\top$ iff some transition $s \xrightarrow{\alpha} s'$ exists in the product. Since the action labels of transitions are not used by the model checking procedure, they are not encoded. The feature expression of a transition is implicitly embedded in the encoding.[3] Formally,

$$\chi_{trans}\big(\bar{s}, \bar{s}', \bar{p}\big) : \{\bot, \top\}^k \times \{\bot, \top\}^k \times \{\bot, \top\}^n \to \{\bot, \top\},$$

such that $\chi_{trans}(enc(s), enc(s'), enc(p)) = \top$ iff some $s \xrightarrow{\alpha} s'$ in $fts_{|p}$. The feature expression on the transition is the cofactor for the encoding of both states:

$$\chi_{\bigvee_\alpha \gamma(s \xrightarrow{\alpha} s')}(\bar{p}) : \{\bot, \top\}^n \to \{\bot, \top\} \triangleq \chi_{trans[\bar{s} \leftarrow enc(s), \bar{s}' \leftarrow enc(s')]}(\bar{p}).$$

Transitions with the same start and end states are implicitly merged (with a disjunction of their Boolean function labels). This yields a symbolic encoding for FTSs covering all of Definition 3, except for actions and action labels.

3.2.2. Symbolic algorithms

Having the fundamentals covered, we can proceed to the model checking algorithms. For now, we consider only CTL formulae. Further in this section, we show how we can adapt our algorithms to extend them to full fCTL. The model checking algorithm for CTL is based on the recursive computation of *satisfaction sets* along the parse tree of the formula. A satisfaction set is a set of states that satisfy a particular sub-formula. A full algorithm for CTL model checking of FTSs is given by the parse-tree computation and a recursive definition of the satisfaction sets.

Satisfaction sets are also encoded by their characteristic function,

$$\chi_{Sat(\phi)}(\bar{s}, \bar{p}) : \{\bot, \top\}^k \times \{\bot, \top\}^n \to \{\bot, \top\},$$

so that $\chi_{Sat(\phi)}(enc(s), enc(p)) = \top$ iff $fts_{|p}, s \models \phi$.

The satisfaction sets for state formulae are rather straightforward:

Definition 9. CTL state formulae satisfaction sets:

$$\chi_{Sat(true)}(\bar{s}, \bar{p}) = \top$$
$$\chi_{Sat(a)}(\bar{s}, \bar{p}) = \chi_a(\bar{s})$$
$$\chi_{Sat(\phi_1 \wedge \phi_2)}(\bar{s}, \bar{p}) = \chi_{Sat(\phi_1)}(\bar{s}, \bar{p}) \wedge \chi_{Sat(\phi_2)}(\bar{s}, \bar{p})$$
$$\chi_{Sat(\neg\phi)}(\bar{s}, \bar{p}) = \neg\chi_{Sat(\phi_1)}(\bar{s}, \bar{p}).$$

Note that at this point we do not distinguish between valid and invalid products. We do this as part of the last step of the algorithm.

To define the satisfaction sets for CTL path formulae, we need to be able to calculate the predecessors of the states satisfying a formula. This is the heart of our fixed point algorithms. All the information it needs is contained in the transition relation, and calculating the predecessors of a single state in a single product amounts to instantiating two arguments of the characteristic function of the transition relation:

$$\chi_{Pre(s,p)}(\bar{x}) : \{\bot, \top\}^k \to \{\bot, \top\} \triangleq \chi_{trans[\bar{s}' \leftarrow enc(s), \bar{p} \leftarrow enc(p)]}(\bar{x}),$$

i.e., the cofactor of the transition relation χ_{trans} for the product p and state s.

[3] Which is natural as both the transitions and the feature expression are Boolean functions.

Of course, this calculation has to be very efficient since it is executed at each step of the algorithm. Therefore, the computation cannot rely on single state/product predecessor computations to accomplish this. We rather need to compute it on *a set* of such couples, generally a satisfaction set of some property ϕ. This leads us to define the operator *SetPre* as follows.

Definition 10.

$$\chi_{SetPre(Sat(\phi))}(\bar{s},\bar{p}) : \{\bot,\top\}^k \times \{\bot,\top\}^n \to \{\bot,\top\} \triangleq \exists \bar{s}' \bullet \chi_{Sat(\phi)}(\bar{s}',\bar{p}) \wedge \chi_{trans}(\bar{s},\bar{s}',\bar{p}).$$

Intuitively, $SetPre(Sat(\phi))$ is the set of couples (s,p) such that there exists a state s' that satisfies ϕ in product p and to which s has a transition in product p. Since the operation is computed on the symbolic encoding of the sets, it does not consider states or products individually.

Given Definition 10, it is straightforward to obtain $Sat(EX\ \phi)$. It is sufficient to calculate the predecessors of $Sat(\phi)$, that is, to apply the *SetPre* operator to $Sat(\phi)$.

Definition 11. $\chi_{Sat(EX\ \phi)}(\bar{s},\bar{p}) \triangleq SetPre(Sat(\phi))(\bar{s},\bar{p})$.

The algorithm for $Sat(E\phi_1 U \phi_2)$ proceeds in the same way as the classical CTL algorithm [7]. It starts with the states and products satisfying ϕ_2 and works backwards, searching for predecessors which satisfy ϕ_1.

Definition 12. $\chi_{Sat(E(\phi_1 U \phi_2))} = \chi_{T_i} \bullet \chi_{T_i} = \chi_{T_{i+1}}$, where

$$\chi_{T_0}(\bar{s},\bar{p}) = \chi_{Sat(\phi_2)}(\bar{s},\bar{p})$$
$$\chi_{T_{i+1}}(\bar{s},\bar{p}) = \chi_{T_i}(\bar{s},\bar{p}) \vee \left(\chi_{Sat(\phi_1)}(\bar{s},\bar{p}) \wedge \chi_{SetPre(T_i)}(\bar{s},\bar{p}) \wedge \neg\chi_{T_i}(\bar{s},\bar{p})\right).$$

In each iteration, we add the states $(\bar{s},\bar{p})$ that satisfy ϕ_1, i.e. $\chi_{Sat(\phi_1)}(\bar{s},\bar{p})$, and are predecessors of a state in T_i, i.e. $\chi_{SetPre(T_i)}(\bar{s},\bar{p})$. An optimisation known in current CTL algorithms, and crucial here, is to only add states that were not already in T_i, i.e. $\neg\chi_{T_i}(\bar{s},\bar{p})$. Otherwise, previously visited states would be re-visited, which would be inefficient due to the added feature variables.

The algorithm for $EG\ \phi$ starts off with all states and products satisfying ϕ and progressively shrinks this set by removing states and products whose successors do not satisfy ϕ.

Definition 13. $\chi_{Sat(EG\ \phi)} = \chi_{T_i} \bullet \chi_{T_i} = \chi_{T_{i+1}}$, where

$$\chi_{T_0}(\bar{s},\bar{p}) = \chi_{Sat(\phi)}(\bar{s},\bar{p})$$
$$\chi_{T_{i+1}}(\bar{s},\bar{p}) = \chi_{T_i}(\bar{s},\bar{p}) \wedge \chi_{SetPre(T_i)}(\bar{s},\bar{p}).$$

The final step of the model checking algorithm is to check whether all initial states satisfy ϕ, and for which products they do. Given $\chi_{Sat(\phi)}(\bar{s},\bar{p})$, the set of products that violate ϕ is obtained by intersecting the complement of $Sat(\phi)$ with the set of initial states, and then projecting on the state variables. This leaves a Boolean function over the feature variables characterising the set of violating products. This set has to be intersected with the set of valid products, unless the calculation was seeded with the valid products.

Definition 14. The set of products $\chi_{px_{bad}}$ violating a CTL property ϕ is

$$\chi_{px_{bad}}(\bar{p}) = \exists \bar{s} \bullet \chi_I(\bar{s}) \wedge \neg\chi_{Sat(\phi)}(\bar{s},\bar{p}) \wedge \mathbb{B}(d)(\bar{p})$$

where $\mathbb{B}(d)$ denotes the Boolean function equivalent to the set of products $[\![d]\!]$ of a feature model d (cf. Definition 1).

If $\chi_{px_{bad}} = \bot$, the property is satisfied by all products.

The algorithms for calculating satisfaction sets combined with the parse tree computation lead to a complete model checking algorithm for CTL, and hence fCTL, over FTSs.

Algorithm 15. Compute $Sat(\phi)$ recursively along the parse tree of ϕ following Definitions 9, 11, 12 and 13. Calculate $\chi_{px_{bad}}$ following Definition 14. If $\chi_{px_{bad}} = \bot$, return $\top$. Otherwise, return $\bot$ and $\chi_{px_{bad}}$.

3.2.3. Reducing fCTL model checking to classical model checking

A closer look at the algorithms for calculating satisfaction sets in the previous section reveals that they do not treat feature and state variables differently. This means that it might be relatively easy to reduce fCTL model checking to classical symbolic model checking. However, in classical symbolic model checking, satisfaction sets only refer to states, not to features. A way to achieve this is to change our symbolic encoding of FTSs slightly, by moving the features from the transitions to the states:

$$\chi_S(\bar{s}, \bar{p}) : \{\bot, \top\}^k \times \{\bot, \top\}^n \to \{\bot, \top\} \bullet \chi_S\big(enc(s), enc(p)\big) = \top \iff s \in S.$$

That is, the features are parameters of the characteristic function of the set of states, but their value does not matter. The initial states and the sets of states that capture the action labelling are defined similarly:

$$\chi_I(\bar{s}, \bar{p}) : \{\bot, \top\}^k \times \{\bot, \top\}^n \to \{\bot, \top\} \bullet \chi_S\big(enc(s), enc(p)\big) = \top \iff s \in I,$$
$$\chi_a(\bar{s}, \bar{p}) : \{\bot, \top\}^k \to \{\bot, \top\} \bullet \chi_a\big(enc(s)\big) = \top \iff a \in L(s).$$

As features are now part of the states, and the symbolic transition relation is a function over two copies of the states, it now has two copies of the features instead of just one:

$$\chi_{trans}\big(\bar{s}, \bar{p}, \bar{s}', \bar{p}'\big) : \{\bot, \top\}^k \times \{\bot, \top\}^n \times \{\bot, \top\}^k \times \{\bot, \top\}^n \to \{\bot, \top\},$$

such that $\chi_{trans}(enc(s), enc(p), enc(s'), enc(p')) = \top$ iff some $s \xrightarrow{\alpha} s'$ in $fts_{|p}$ and $p = p'$. This version of the symbolic transition relation is only equivalent to the one of Section 3.2.1 if the feature variables are left unchanged by the transition relation. This is ensured by the last condition, $p = p'$.

Given that features are now part of the states, the characteristic function of the satisfaction sets keeps the same signature as before and the predecessor calculation becomes:

Definition 16. $\chi_{SetPre(Sat(\phi))}(\bar{s}, \bar{p}) \triangleq \exists \bar{s}' \bullet \chi_{Sat(\phi)}(\bar{s}', \bar{p}) \wedge \chi_{trans}(\bar{s}, \bar{p}, \bar{s}', \bar{p})$.

This definition coincides with the definition of the predecessors in standard symbolic CTL model checking algorithms, under the condition that the feature variables in a transition do not change. Furthermore, satisfaction sets in our algorithm now coincide with those in standard symbolic CTL model checking algorithms, and even their calculation is the same. By combining these observations we can, at least for the calculation of $Sat(\phi)$, reduce symbolic FTS model checking to symbolic model checking of specially crafted transition systems.

It is in the final step of the algorithm, i.e., checking whether all initial states satisfy the property, where feature variables are treated differently from the states. This step needs to be adapted as described in Definition 14, by quantifying away the state variables in order to obtain the set of violating products. If this is not done, the algorithm will just yield $\bot$ if there are violating products, without indicating which products are to blame.

The modified encoding has another advantage: it allows to express fCTL properties in CTL. This is due to the fact that feature variables are state variables that can be referenced in a property. A set of products χ_{px} can thus be expressed in the specification language, which means that the fCTL formula $[\chi_{px}]\phi$ can be translated to the CTL formula $(\chi_{px}) \Longrightarrow \phi$.

3.2.4. Time complexity

For reference, single-system CTL model checking of transition systems has a computational complexity of $O(|S|.|\phi|)$. For FTSs and fCTL, an enumerative algorithm that iterates through the set of products ($O(2^n)$), calculates their projection ($O(|expr|)$) and model checks the resulting transition systems ($O(|\phi||fts|)$) would yield a total time complexity of $O(2^n|\phi||fts|)$.

The algorithmic complexity of Algorithm 15 for FTS CTL model checking is $O(|fts|.|\phi|.2^n)$. Basically, a satisfaction set is calculated for each node in the formula giving the factor $|\phi|$. This calculation is linear in the size of the state space for $Sat(\top)$, $Sat(a)$, $Sat(\neg\phi)$, $Sat(\phi_1 \wedge \phi_2)$ and $Sat(EX\ \phi)$. The fixed points of $Sat(E(\phi_1 U \phi_2))$ and $Sat(EG\ \phi)$ both take $O(|S|.2^n)$ since, in the worst case, they proceed monotonically through 2^n products for each state.

With regards to computational complexity, an enumerative algorithm that would check the TS of each product individually is thus equal to ours. This is consistent with the fact that FTS CTL model checking can be reduced to the classical symbolic CTL model checking algorithm, used by the naïve algorithm. The difference between both is that the naïve algorithm performs $O(2^n)$ model checks of models of size $O(|fts|)$, whereas our algorithm performs a single model check, of a model of size $O(2^n|fts|)$. Our hope is that the similarities between the $O(2^n)$ models will cause the BDDs of the single model to be smaller than the sum of the size of the smaller BDDs in the naïve algorithm. Furthermore, as variable orderings play a crucial role in the efficiency of BDD operations, our algorithm has the advantage of requiring only a single ordering (which can then be tuned).

Note that the additional exponential factor of our algorithm cannot be avoided unless model checking is restricted to models less powerful than FTSs, as done in [33]. Also note that our algorithm remains linear in the size of the state space and is more efficient than the one presented in [32]. More precisely, the latter is $O(|\phi|.|S|!) = O(|\phi|.|S|^{|S|})$, and the models

it treats can be transformed to FTSs in time linear with the size of the models. The algorithm of [32] is thus in EXPTIME whereas ours is in E, i.e. DTIME($2^{O(x)}$), a class that "*captures a more benign aspect of exponential time*" [35]. Furthermore, it is important to note that in practice, the size of the state space is much larger than the number of features. Indeed, the state space is defined by a set of variables, which means that its size is exponential in the number of variables. Since the number of variables is likely to be larger than the number of features, the size of the state space is likely to be much larger than 2^n, which would mean that $O(|S|) = O(2^n|S|)$. In practice, the CTL model checking problem for FTSs is thus not inherently harder than the one for transition systems.

4. The fSMV language

As we have shown, the fully symbolic model checking algorithm for FTSs can largely be reduced to the classical algorithm for transition systems. To implement it, we decided to extend the state-of-the-art symbolic model checker NuSMV [6].[4] Thereby, we can take advantage of its existing infrastructure. We refer to the extended NuSMV as 'fNuSMV'.

The input language of NuSMV can be used as-is to create models that correspond to the symbolic encoding required for FTS model checking. However, to make modelling more intuitive, we reuse a language by Plath and Ryan [36], which is specifically designed for specifying features. The language, which we call 'fSMV', is a feature-oriented extension of the input language of NuSMV.[5] It is based on the compositional feature-oriented software development paradigm: features are specified independently, and a product is created by composition.

4.1. Syntax

Essentially, a NuSMV model consists of a set of variable declarations and a set of assignments. The variable declarations define the state space and the assignments define the transition relation. In each assignment, the value of a variable in the next state is defined in function of the variable values in the present state. For each variable, there can also be an assignment that defines its initial value. Alternatively, the value of a variable can be defined directly in function of the other variables. Modules can be used to encapsulate and factor out recurring elements. Henceforth, we will refer to this language as 'SMV'.

The typical example of an SMV model (taken from [5]) is the following.

```
MODULE main
VAR
  request: boolean;
  state: {idle, busy};
ASSIGN
  init(state) := idle;
  next(state) := case state = idle & request: busy;
                      true: {idle, busy};
                 esac;
```

The above model describes a controller that is either idle or busy treating a request. The `VAR` section defines variables, and the `ASSIGN` section defines their values (and thereby the transition relation). Requests are modelled by the variable `request`. The absence of any assignments for this variable means that its value is chosen non-deterministically in each state, which models the fact that requests are controlled by the environment. The `state` variable represents the state of the controller. It is of an enumerated type. The `init` assignment defines its initial value (initially, the system is idle). The `next` assignment, defines the transition relation: when the controller is idle and there is a request, it will treat the request and be busy (line 7), otherwise, it may continue to be busy for a while and return to idle once the request is treated (line 8). A `case` statement is a conditional expression where each line is of the form `condition: value;`. The conditions are evaluated in the order in which they are specified, and the value of the first true condition is taken. The `1` at line 8 means *true*, i.e., it acts like an 'else' in programming languages such as C or Java. The `{idle, busy};` at line 8 is the non-deterministic choice between those values.

The semantics of such a model is a transition system. Its state space is the product of the domains of the variables. The state space can be narrowed with the keyword `INVAR`, which is used to define a constraint which has to hold in all states. The initial states of the transition system are those whose variable values satisfy the `init` statements. Similarly, there is a transition between a pair of states when their variable values satisfy the `next` statements. SMV has other ways to define the transition relation. In this work, we restrict ourselves to the `ASSIGN` syntax. In addition to the transition system, an SMV model also contains CTL properties. These are called 'specifications' and follow the `SPEC` keyword.

Of course, NuSMV never constructs this transition system explicitly. Instead, it constructs a symbolic transition relation, i.e., a Boolean function with two copies of each variable, one for the start state and one for the end state. This Boolean

[4] http://nusmv.irst.itc.it.

[5] More precisely, fSMV extends the input language of the earlier SMV model checker. The input language of NuSMV is almost identical.

function can be derived almost immediately from the assignment statements: when `var` refers to a variable in the start state, `next(var)` refers to the variable in the end state. The Boolean function is thus the conjunction of all assignment statements.

An fSMV model consists of a base system, such as the one shown above, a list of features and an FD in TVL [10] syntax.[6] A base system is specified in SMV, whereas features are specified in a dialect of SMV. The whole language is called 'fSMV'. Features in fSMV are based on *superimposition* [26]: a feature describes the changes to be made to the base system. A feature declaration consists of three parts [36]:

(1) `REQUIRE` defines variables that the feature needs. These have to be defined in the base system, or by other features. The `REQUIRE` clauses define constraints on the order in which features can be composed.
(2) `INTRODUCE` defines new variables or specifications that the feature adds to the system.
(3) `CHANGE` defines changes made to existing variables. Types of change are:
  (3.1) `IMPOSE` a new definition of an existing variable. This means that the feature replaces the `init` or `next` state definition of the variable. An `IMPOSE` clause can be guarded with an `IF` clause, meaning that it only has an effect if a certain condition holds.
  (3.2) `TREAT` existing variables differently. When the value of the variable is read inside the definition of some other variable, the read value is modified. `TREAT` clauses can also be guarded, but this is syntactic sugar [36].

As an example, consider a feature *Sleep* which adds a switch to the system that causes it to discard any further request. The switch is modelled with a new non-deterministic variable `sleep` (using `INTRODUCE`). The system is changed in such a way that if the system is sleeping and finished treating requests, then it will stay idle, not accepting any new requests (using `IMPOSE`).

```
FEATURE sleep

REQUIRE
  MODULE main
  VAR state: {idle, busy};

INTRODUCE
  MODULE main
  VAR sleep: boolean;

CHANGE
  MODULE main
  IF sleep & state = idle THEN
     IMPOSE next(state) := idle;
```

Given a base system and a feature whose `REQUIRE` constraints are satisfied by the base system, the *feature composition* operation creates a new base system. Feature composition is syntactic, and consists in replacing existing `assign` or `init` statements, and adding new variables. It is performed in three steps: first, `TREAT` assignments are applied, then `IMPOSE` and then `INTRODUCE` assignments.

The composition of the base system and the preceding *Sleep* feature yields the following system.

```
MODULE main
VAR
  request: boolean;
  state: {idle, busy};
  sleep: boolean;
ASSIGN
  init(state) := ready;
  next(state) :=
    case sleep & state = idle: idle;
         true: case state = idle & request: busy;
                    true: {idle, busy};
            esac;
         esac;
```

To make it possible to structure large models and to reuse model fragments, SMV and fSMV have the `MODULE` syntax. A module encapsulates variables and assignments and can be used as a type inside other modules. The `main` module defines the behaviour of the system. Any other module must be used in the `main` module (or in a module used in the `main`

[6] Note, however, that FDs are not part of the original definition in [36] and are currently not implemented by our toolset.

module). Modules can be parameterised. A parameter is a reference to a variable to which the module would otherwise not have access. Modules can be considered syntactical sugar and can easily be eliminated by a syntactic procedure. When we give formal definitions, we thus abstract away from modules.

4.2. Semantics

As described above, the semantics of a normal SMV model (or an fSMV base system) is a transition system. Furthermore, as the composition of a base system and a feature yields another base system, the semantics of a product (i.e., a base system composed with several features) is a transition system as well. This is consistent with FTSs, where the behaviour of a product is also a transition system. However, if fSMV is to serve as a high-level language for FTSs, we need to express its semantics in terms of FTSs. We do this by giving a translation from fSMV to symbolic FTSs as defined in the previous section. To be able to do this in a precise manner, we need to formalise the description of fSMV.

Let us start by defining an fSMV base system.

Definition 17. Let V be a set of variables, D a set of (finite) domains or types, and $E(V)$ the set of all SMV expressions over V. Let $A(V_1, V_2)$ be the set of assignments where variables in V_1 can be on the left-hand side and expressions over variables in V_2 can be on the right-hand side. Formally, $A(V_1, V_2) \subseteq \{-, \texttt{init}, \texttt{next}\} \times V_1 \times E(V_2)$, that is, a set of triples (s, d, e) where s distinguishes between `v` (the `-`), `init(v)` or `next(v)` for $v \in V_1$, and $e \in E(V_2)$ is an expression.[7] A base system m is a tuple $m = (v, \tau, a, p)$, where

- $v \subseteq V$ is a set of variables,
- $\tau : V \to D$ is a function assigning a domain to each variable,
- $a \subseteq A(v, v)$ is a set of assignments, and
- $p \subseteq \mathcal{P}(v) \times \mathcal{P}(v)$ is a (possibly empty) set of processes. A process is a couple (v_p, w_p) where v_p denotes the set of variables read by the process and $w_p \subseteq v_p$ denotes the set of variables written by the process. Furthermore, SMV requires that the sets of variables written by different processes do not overlap.

For a model without parallel composition, the set p is empty. The semantics of a base system is a transition system [34].

As said before, for the purpose of this discussion, we abstract away from modules without loss of generality. Using these notions the base system shown earlier corresponds to the tuple (c, τ, a, p) with $v = \{request, state\}$, $\tau = \{(request, \{\bot, \top\}), (state, \{idle, busy\})\}$, $p = \emptyset$, and

$$a = \big\{(init, state, idle), (next, state, \texttt{case...esac;})\big\}.$$

An fSMV model is defined as follows.

Definition 18. An fSMV model is a triple (b, d, G), where b is a base model as defined in Definition 17, d is an FD as defined in Definition 1 and G an ordered list of implemented features. Let N be the set of features in the FD, we assume there to be a bijective function $impl : N \to G$ with codomain G, that associates features from the FD and their *implementations* in the model. When it is clear from the context, we write f instead of $impl(f)$ or $impl^{-1}(f)$. Each feature $f \in G$ is a tuple consisting of

- $v_f \subseteq V$, a set of new variables;
- $\tau_f : v_f \to D$, a type function;
- $p_f : p \to \mathcal{P}(v_f) \times \mathcal{P}(v_f)$, a function that tells for each process whether the new variables belong to it (read and write respectively); sets of written variables cannot overlap;
- $a_f \subseteq A(v_f, v \cup v_f)$, a set of `INTRODUCE` assignments;
- $m_f \subseteq E(v \cup v_f) \times A(v, v \cup v_f)$, a set of guarded `IMPOSE` assignments. The first element is the guard, the second is an assignment where the left-hand side is the variable that is affected, and the right-hand side the value it takes if the guard is true;
- $t_f \subseteq A(v, v \cup v_f)$, a set of `TREAT` assignments. The left-hand side is the variable that is affected, and the right-hand side the value substituted by the feature.

This definition does not formalise the `REQUIRES` constraint which has no effect on the behaviour of the products. Accordingly, the *sleep* feature given earlier is formally represented by the tuple

$$\big(\{sleep\}, \big\{\big(sleep, \{\bot, \top\}\big)\big\}, \emptyset, \emptyset, \big\{\big(\texttt{sleep = true\&state = idle}, (\texttt{next}, \texttt{state}, \texttt{idle})\big)\big\}, \emptyset\big).$$

[7] An expression alone is not an assignment, it just defines a value.

Since feature composition is a syntactical operation, composing features in a different order might lead to a different result, i.e., two features do not necessarily commute. Intuitively, features composed later can override changes made by earlier features. Strictly speaking, this means that a product in fSMV is a *list* of features, not a set. This is incompatible with the way products are defined in FTSs and FDs. A way around this mismatch would be to assume that condition IV of [36] is met: that the order of features is irrelevant, i.e., all features commute: $(b \otimes f_i) \otimes f_j = (b \otimes f_j) \otimes f_i$ for any base system b, features f_i and f_j, where $\otimes$ is the composition operator defined below. This assumption would allow us to consider a product as a set of features. However, it would also exclude any model in which two features change the same variable. We thus opted for a different solution, which is to assume that a total ordering of the features is given as part of the model. With this assumption, a product can also be given by a set of features. Note that this drastically reduces the number of products, from $O(\sum_{i=0}^{n} \frac{n!}{(n-i)!})$ to $O(2^n)$.

Feature composition can be formally defined as follows.

Definition 19. Composition of a base system $b = (v, \tau, a, p)$ and a feature $f = (v_f, \tau_f, p_f, a_f, m_f, t_f)$ is noted $b \otimes f$ and produces a new base system $b' = (v', \tau', a', p')$, where

- $v' = v \cup v_f$ and $\tau' = \tau \cup \tau_f$.
- a' is obtained by first applying t_f to a, then m_f, and finally adding a_f, formally:

$$
\begin{aligned}
a' &= a_m \cup a_f \\
a_m &= \big\{(s, d, e') \mid (s, d, e) \in a_t \wedge \\
&\qquad \text{if } \exists\big(g, (s', d', e'')\big) \in m_f \bullet s = s' \wedge d = d' \\
&\qquad \text{then } e' = \texttt{case } g : e''; \ \mathit{true} : e \ \texttt{esac}; \\
&\qquad \text{else } e' = e\big\} \\
a_t &= \big\{(s, d, e') \mid (s, d, e) \in a \wedge e' = \mathit{treat}(e, t_f)\big\}
\end{aligned}
$$

  where $\mathit{treat}(e, t_f)$ transforms e so that for all $(s, d, e'') \in t_f$, the occurrences of $s(d)$ are replaced by e''.
- $p' = \{(v \cup v_f, w \cup w_f) \mid (v, w) \in p \wedge p_f(v, w) = (v_f, w_f)\}$.

The definition of a', the set of assignments of the composed system, is somewhat cryptic. It is defined with three intermediate results. The first is a_t, i.e., after the `TREAT` assignments were applied. A `TREAT` assignment changes the right-hand side of all assignments by replacing the occurrences of a variable by an expression. The second intermediate result is a_m, i.e., after `TREAT` and `IMPOSE` were applied. An `IMPOSE` assignment replaces the right-hand side of a single assignment (the variable it concerns) by a case statement: if the guard is true, the replacement expression is used, otherwise, the previous expression.

Note that the definition of a_t may lead to errors when there exists a `TREAT` assignment $(--, d, e)$ such that e is non-deterministic, that is, e is evaluated to a set of values instead of a single value. To avoid these errors, e must first be transformed into the union of deterministic expressions, that is, one per value of e (see [36] for details).

We intentionally keep these definitions at a high level of abstraction. They are sufficiently detailed to make the following discussion precise and abstract enough to make it intuitive. In particular, we do not detail the syntax or semantics of expressions and types. There are a number of rules on what constitutes a valid model (w.r.t. types, variable names, etc.) which we also omit. The interested reader is referred to [34,5,37] for a detailed formal definition of SMV, NuSMV and fSMV.

4.3. From fSMV to FTSs and back

As said before, given a base system b and a list of features, $b \otimes f_1 \otimes \cdots \otimes f_n$ denotes a symbolic transition system. To produce a symbolic FTS, we can use the *lifting* technique of [38]. The idea is to introduce a new Boolean variable for each feature. Furthermore, all changes made by a feature are guarded by its feature variable. This leads us to define *lifted* feature composition as follows (the changes w.r.t. Definition 19 are shown in colour).

Definition 20. Lifted composition of a base system $b = (v, \tau, a, p)$ and a feature $f = (v_f, \tau_f, p_f, a_f, m_f, t_f)$ is noted $b \odot f$ and produces a new base system $b' = (v', \tau', a', p')$, where

- $v' = v \cup v_f \cup \{\mathit{var}(f)\}$ and $\tau' = \tau \cup \tau_f \cup \{\mathit{var}(f), \{\bot, \top\}\}$, where $\mathit{var}(f)$ denotes the feature variable associated to f.
- a' is obtained by first applying t_f to a, then m_f, adding a_f, and finally an assignment that requires the feature variable to remain constant formally:

$$a' = a_m \cup a_f \cup \big\{\big(\texttt{next}, \mathit{var}(f), \mathit{var}(f)\big)\big\}$$

$$
\begin{aligned}
a_m \;=\; & \big\{(s,d,e') \;\big|\; (s,d,e) \in a_t \wedge \\
& \quad \text{if } \exists\big(g,(s',d',e'')\big) \in m_f \bullet s = s' \wedge d = d' \\
& \quad \text{then } e' = \texttt{case}\ \mathit{var}(f)\ \&\ g : e''\texttt{;}\ \mathit{true} : e\ \texttt{esac;} \\
& \quad \text{else } e' = e\big\} \\
a_t = \big\{(s,d,e') \;\big|\; & (s,d,e) \in a \wedge e' = \mathit{treat}(e,t_f)\big\}
\end{aligned}
$$

where $\mathit{treat}(e,t_f)$ transforms e so that for all $(s,d,e') \in t_f$, the occurrences of $s(d)$ are replaced by `case` $\mathit{var}(f)$: e'; $\mathit{true} : e$ `esac;`.

- $p' = \{(v \cup v_f, w \cup w_f) \mid (v,w) \in p \wedge p_f(v,w) = (v_f, w_f)\}$.

Guarding a change made by a feature with the corresponding feature variable means that the behaviour *with* (resp. *without*) the feature can be obtained by setting the feature variable to $\top$ (resp. $\bot$). For example, the lifted composition of the controller base system and the *Sleep* feature yields the following.

```
MODULE main
VAR
  sleepFeature: boolean; -- added feature variable
  request: boolean;
  state: {idle, busy};
  sleep: boolean;
ASSIGN
  next(sleepFeature) := boolean;
  init(state) := idle;
  next(state) :=
    case sleepFeature & sleep & state = idle: idle;
         true: case state = idle & request: busy;
                    true: {idle, busy};
            esac;
    esac;
```

The lifted composition of a base system and a list of features, $b \odot f_1 \odot \cdots \odot f_n$ denotes a symbolic FTS. It corresponds to the symbolic encoding given in Section 3.2.3: the features are part of the states because each has a Boolean feature variable, the feature variables are initialised non-deterministically, and they do not change their value as part of a transition. Since this lifted composition is also a valid SMV model, and given that symbolic FTS model checking can be reduced to model checking of symbolic transition systems, we can feed it as-is into NuSMV and reuse the result of the satisfaction set computation.

Recall that an fSMV model is a base system and a *list* of features. The lifted composition of the base system and all features in the given order is thus always well-defined. To simplify the notation, we therefore write $\mathit{fts}(m) \triangleq b \odot f_1 \odot \cdots \odot f_n$, for an fSMV model $m = (b, d, (f_1, \ldots, f_n))$, to denote the corresponding symbolic FTS. A symbolic FTS can be projected to a product by fixing the values of the feature variables according to the product:

Definition 21. Given an fSMV model m with features G, the projection of $\mathit{fts}(m) = (v, \tau, a, p)$ to a product $p \in [\![d]\!]_{FD}$ is the model $\mathit{fts}(m)_{|p} \triangleq (v, \tau, a', p)$ where $a' = a \cup \{(\mathtt{init}, \mathit{var}(f), f \in p) \mid f \in G\}$.

We have shown that lifted feature composition indeed yields a symbolic FTS. What is left to be shown is that it preserves the semantics of the normal feature composition as it was defined by [36]. This is indeed the case. The following theorem establishes that projecting the fSMV obtained by lifted composition (Definition 20) to a product leads to an SMV model that is syntactically equivalent to the one obtained by normal feature composition (Definition 19), with one small exception. The projection of a lifted composition can have more variables than the normal composition: those added by non-selected features. Since these are not removed by projection, they remain in the projected system. However, they do not influence the other variables, which means that the projected system and the one obtained by normal feature composition are *bisimilar* [4] when only considering common variables, thus preserving any CTL properties over these variables.

Theorem 22. *For any fSMV model $m = (b, d, \{f_1, \ldots, f_n\})$ and product $p = \{f_i, \ldots, f_j\}$, where the indexing from i to j corresponds to the feature order given in m,*

$$b \otimes f_i \otimes \cdots \otimes f_j \sim (b \odot f_1 \odot \cdots \odot f_n)_{|p}$$

where $\sim$ denotes bisimilarity w.r.t. shared variables.

Proof. First, observe that lifted feature composition does not remove any of the code in the base system; rather, it wraps changed code inside `case` statements. In a projection, the values of the features are all fixed. This means that references to features can be replaced everywhere by their value. Any changes by non-selected features will be of the kind

`case` *false* & $\ldots : exp_1$; *true* : exp_2 `esac;`, where exp_2 is the expression that was there before the feature was added. Such `case` statements can be simplified to exp_2 ;. For selected features, the simplification is similar, except that the change made by the feature is preserved. Intuitively, these semantics-preserving transformations correspond to unwrapping of `case` statements introduced by lifted composition. The result is syntactically equivalent to $b \otimes f_i \otimes \cdots \otimes f_j$, with the exception of added variables (and their assignments), which do not influence the common variables. Hence, there is a bisimulation relation between two SMV models, when only considering common variables. These are the only variables over which properties to be preserved have to hold. □

We define the semantics of fSMV in terms of SMV, i.e., as a symbolic FTS. In this case, the parallel composition is defined as it is for SMV. An alternative would be to define the semantics in terms of explicit FTSs (Definition 3). This would make it possible to define the semantics of fSMV compositionally, where each process translates to an FTS. We have done this in [8].

The above results establish that any fSMV can be translated to an FTS, i.e., that the fSMV language is a subset of the FTS language. The following theorem establishes that the converse also holds, i.e., that both languages are expressively equivalent.

Theorem 23. *Any FTS can be translated into fSMV.*

Proof. Given an FTS $(S, Act, trans, I, AP, L, d, \gamma)$, construct an fSMV model (b, d, F) with $b = (v, \tau, a, \emptyset)$, where

(1) one variable of the fSMV, *state*, is used to encode all the states of the FTS: $\tau(state) = S$. For every feature $f \in N$, the fSMV will have a variable f with $\tau(f) = \{\bot, \top\}$. Hence, $v = \{state\} \cup N$;
(2) in the base system, the feature variables are always $\bot$. The assignments related to the feature variables are thus $a_F = \{(-, f, \bot) \mid f \in N\}$. The initial value of the *state* variable is the non-deterministic choice between the initial states of the FTS, and its `next` value is derived from the transition relation of the FTS. The assignments for the *state* variable are

$$a_s = \left\{(\texttt{init}, state, I), (\texttt{next}, state, \texttt{case}\ case_1 \ldots case_k\ \texttt{esac;})\right\}$$

where the $case_i$ are given by $\{state = s\ \&\ \gamma(s \xrightarrow{\alpha} s')\text{: } s';\ \mid s \xrightarrow{\alpha} s' \in trans\}$. The set of assignments is then $a = a_F \cup a_s$;
(3) each feature imposes that its associated variable (which is part of the base system) takes the value $\top$, i.e.,

$$F = \left\{(\emptyset, \emptyset, \emptyset, \emptyset, \left\{\left(\top, (-, f, \top)\right)\right\}, \emptyset) \;\middle|\; f \in N\right\}.$$

In consequence, the composition of a base system with a set of features yields a transition system of which all transitions whose $\gamma(s \xrightarrow{\alpha} s')$ evaluates to $\bot$ for the feature variables have been removed. This corresponds exactly to projection as defined in Definition 4. □

5. Implementation in NuSMV

We now give an overview of our model checking toolset. First, we present it from the user perspective. We then discuss more technical details of the implementation.

5.1. User interface and illustration

Like NuSMV, our toolset is command-line based. This is rather natural, especially since the composition scripts lend themselves well to pipe-based chaining of commands.

The input to the tool is a model expressed in fSMV. Concretely, this means at least one file containing the base system and one file for each feature. The feature order is not explicitly part of the syntax, it is specified when using the composition script. The composition script, `compose.php`, is indeed the first tool to be used in a normal use case. Let us illustrate this with the fSMV model presented in the previous sections. Assume that there are two files `base.smv` corresponding to the base system and `sleep.feat` corresponding to the *sleep* feature. The feature composition (following Definition 19) of the base system and the *Sleep* feature is given by:

```
$ php compose.php sleep.feat < base.smv > ts.smv
```

The composition script is written in PHP, which is why the command starts with `php`. The result, `ts.smv`, is the behaviour of the product consisting of the base system and the sleep feature. It can be analysed using NuSMV. To produce the lifted feature composition (Definition 20) instead of the normal feature composition, the command-line parameter `-l` has to be set, i.e.:

```
$ php compose.php -l sleep.feat < base.smv > fts.smv
```

The script thus has one parameter, the file name of the feature to be composed, and reads the base system from `stdin` (hence the input redirection < in the previous two examples). This means that when there are several features to compose, the calls can be chained with pipes, e.g.:

```
$   php compose.php -l sleep.feat < base.smv
  | php compose.php -l other.feat > fts.smv
```

When chaining multiple calls, the `-l` parameter has to be set for all of them or for none of them.

The heart of our toolset is fNuSMV, which is a modified version of NuSMV. It is distributed as a patch (created using the standard Unix `diff` tool) which has to be applied to the NuSMV codebase before NuSMV is compiled. When this is done, a new command-line parameter `-fbdd` is available in NuSMV.

Let us illustrate the use of fNuSMV with the controller example. A property for such a system would be that "*every request eventually results in the controller being busy*". In CTL, this becomes $\forall G\ (request \Rightarrow \forall F\ state = busy)$, or in NuSMV syntax `SPEC AG (request -> AF state=busy)`. In NuSMV, the temporal operators are written with letters: *X* becomes `X`, *G* becomes `G` and *F* becomes `F` (the `U` for *until* remains). Checking this property using fNuSMV would look as follows.

```
$ ./NuSMV -fbdd fts.smv
*** This is NuSMV 2.5.0 [...]
-- Computing fbdd init.. done.
-- specification AG (request -> AF state = busy)  is false
-- (fbdd) specification is false for products satisfying:
   f.fSleep
-- (fbdd) specification is true for products satisfying:
   !f.fSleep
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  f.fSleep = TRUE
  request = FALSE
  state = idle
  sleep = FALSE
-- Loop starts here
-> State: 1.2 <-
  request = TRUE
  sleep = TRUE
-> State: 1.3 <-
```

The global result, printed at line 4, is that the property is violated. At line 6, fNuSMV prints a feature expression characterising the products that violate the property, in this case all those containing the *Sleep* feature. At line 8, the negation of this feature expression is shown (this helps in the case of larger expressions). This is followed by a counterexample at lines 10 to 21. This counterexample only applies to a single one of the violating products (identified by the value assignments to feature variables in the first state). This is a small drawback compared to our LTL model checking algorithm, which computes a counterexample that holds for all violating products. Development of such a technique for CTL counterexamples is part of our future work.

As a comparison, if we use NuSMV without the `-fbdd` parameter, the result will be the following.

```
$ ./NuSMV fts.smv
*** This is NuSMV 2.5.0 [...]
-- specification AG (request -> AF state = busy)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  f.fSleep = TRUE
  request = FALSE
  state = idle
  sleep = FALSE
-- Loop starts here
-> State: 1.2 <-
  request = TRUE
  sleep = TRUE
-> State: 1.3 <-
```

This procedure identifies only a single violating product, given by the initial values of the feature variables in the counterexample (line 8). The question whether other products violate or satisfy the property is left open by this check.

In addition to `compose.php` and fNuSMV, our toolset comprises a quantification script, `quantify.php`. This script allows to create parameterised models. In the case of the controller, for example, one could imagine that the controller is

capable of treating several requests in parallel. The parameter of the model would be the number of requests. For the value two, the model would be the following:

```
MODULE main
VAR
  request1: boolean;
  request2: boolean;
  state: {idle, busy};
ASSIGN
  init(state) := idle;
  next(state) :=
    case state = idle & request1 & request2: busy;
         true: {idle, busy};
esac;
```

In the SMV (and thus fSMV) syntax, it is not possible to specify the model for a number of requests n. This capability is added with the quantification script which acts as a preprocessor. Inside a preprocessed model file, the syntax `[forall p=i..j]body[/forall]` can be used:

`i, j` are integers that define a range. Instead of an integer, the letter `n` can be used, which will be substituted by a user-defined value when running the script. The bounds can also be given with expressions, e.g. `p=n-1..n`.
`body` is the piece of text to be repeated.
`p` is the name of the index. It can be referenced inside `body` by writing `%p%` (the index enclosed by percent signs). Simple expressions such as `%p-1%` are also supported.

In the case of the controller example, this would be used as follows.

```
MODULE main
VAR
  [forall i=1..n]
    request%i%: boolean
  [/forall]
  state: {idle, busy};
ASSIGN
  init(state) := idle;
  next(state) :=
    case state = idle [forall i=1..n] & request%i%
                      [/forall]: busy;
         true: {idle, busy};
esac;
```

To produce the example with two requests, the quantification script is used to process the above file.

```
$ php quantify.php 2 base.smv > base-2requests.smv
```

Quantifiers have to be instantiated before a file can be used in the composition script. This can make the use of the tools rather cumbersome. In practice, it is helpful to encapsulate all quantification and composition commands in a shell script. Moreover, most models do not require quantifiers.

5.2. Implementing composition

The composition script implements feature composition and lifted feature composition as specified in Definitions 19 and 20. It is slightly more involved than the definitions suggest, mainly because it has to deal with modules. In both cases, the composition script has to respect the scope of the module to which a change is applied. Moreover, for lifted feature composition, the composition script has to add feature variables so that they can be referenced in all modules uniformly. NuSMV does not allow access to variables in parent modules. This is only possible by parameterising modules. To make the feature variables available in all modules, the composition script therefore creates a new module with all feature variables, instantiates it in the `main` module, and adds it as a parameter to all other modules.

The module containing the features is called `features`, which means that 'features' cannot be used as an identifier in the fSMV files. To each feature corresponds one variable in this module, the variable name being the feature name (with the first letter in uppercase) prefixed by the letter `f`. The `feature` module is declared in the `main` module as a variable named `f`. Therefore, `f` is also a reserved keyword which should not be used inside fSMV models. The parameter added to all other modules is also called `f`, so that the features can be referenced the same way inside the whole model.

For the controller example, the lifted feature composition would result in the following SMV model.

```
MODULE features
VAR
  fSleep: boolean;
ASSIGN
  init(fSleep) := {false,true};
  next(fSleep) := fSleep;

MODULE main
VAR
  f: features;
  request: boolean;
  state: {idle, busy};
  sleep: boolean;
ASSIGN
  next(state) :=
  case f.fSleep & sleep & state = idle: idle;
       true: case state = idle & request: busy;
                  true: {idle, busy};
          esac;
esac;
```

Note that the feature variables are initialised non-deterministically and that their value is defined to be constant.

Naming conventions allow us to easily distinguish feature variables from other Boolean variables. All feature variables have the prefix `f.f`: the first `f` identifies the variable of the main module that holds the feature module and the second `f` is the one prefixed to every feature variable. We need to be able to distinguish feature variables from the other variables when calculating the products for which a certain property holds. An alternative to the naming convention would have been to extend the SMV language. We chose a naming convention as this necessitates far less changes to the NuSMV codebase.

Furthermore, the naming convention means that fCTL properties can be written and attached to any module in the same way. E.g., in the case of the controller, the property that every request will eventually result in a busy controller might not be relevant for controllers with the *Sleep* feature. In fCTL, this can be written $[!sleep]AG\ request \Rightarrow AF\ state = busy$. In SMV, this becomes `SPEC !f.fSleep -> AG (request -> AF state=busy)`. As expected, the property is satisfied by all products:

```
$ ./NuSMV -fbdd fts.smv
*** This is NuSMV 2.5.0 [...]
-- Computing fbdd init.. done.
-- specification (!f.fSleep -> AG (request -> AF state = busy))
is true
```

5.3. Implementing FTS model checking

The output of the composition tool is a normal SMV model and can be model checked directly by NuSMV. However, as shown before, standard NuSMV model checking does not fully exploit the feature encoding. Since NuSMV executes the standard CTL model checking algorithm, it will report *false* if it finds a counterexample. More precisely, it will return *false* if just one of the products violates the property.

Basically, given a property ϕ, the algorithm will compute a Boolean function $\chi_{Sat(\phi)}(\bar{s}, \bar{p})$, where $\bar{s}$ (resp. $\bar{p}$) is the Boolean encoding of some state (resp. some product). $\chi_{Sat(\phi)}$ is true for all states and products that satisfy the property. The normal model checking algorithm will just check whether there exists some initial state for which $\chi_{Sat(\phi)}(\bar{s}, \bar{p})$ is *false*. Unable to distinguish between feature variables (belonging to $\bar{p}$) and normal variables (belonging to $\bar{s}$), the test will existentially quantify over the feature variables which corresponds to considering a single product only.

As discussed in Section 3.2.3, there is sufficient information to determine exactly which products violate and which satisfy the property. The idea is to only quantify $\chi_{Sat(\phi)}$ existentially over the state variables (i.e., those that do not represent features). The result is a Boolean function over the feature variables that represents exactly the products for which the property holds. Implementing this calculation is the only significant change we made to the NuSMV code (the other being the addition of the command-line parameter). Still, it accounts for just about 44 lines of additional code. One fragment of code is executed once for all properties in the model, it consists in creating the BDDs of the feature variables which are used in the quantification. The second fragment is executed for each property, after the model checking algorithm is finished. It calculates the quantifications and prints the information about satisfying and violating products on `stdout`.

Currently, FDs are not implemented as part of the toolset. However, this would be quite straightforward. Following Definition 14, it would be sufficient to test whether the conjunction of the returned function and $\mathbb{B}(d)$, the Boolean function equivalent of the FD d, is satisfiable. An alternative would be to limit the check to the set of valid products of the FD from the outset. This can be done by adding constraint `IVAR` $\mathbb{B}(d)$ to the model before it is analysed. NuSMV considers it as an invariant, which will effectively prevent it from considering invalid products.

Table 1
Benchmarked properties.

ID	Property
01	$AG\ (landingBut2.pressed \Rightarrow AF\ (lift.floor = 2 \wedge lift.door = open))$
01′	$\neg AG\ (landingBut2.pressed \Rightarrow AF\ (lift.floor = 2 \wedge lift.door = open \wedge lift.direction = down))$
02	$AG\ (liftBut3.pressed \Rightarrow AF\ (floor = 3 \wedge door = open))$
03a	$AG\ (floor = 2 \wedge liftBut6.pressed \wedge direction = up \Rightarrow A[direction = up U floor = 6])$
03b	$AG\ (floor = 6 \wedge liftBut1.pressed \wedge direction = down \Rightarrow A[direction = down U floor = 1])$
04	$\neg AG\ (door = closed \Rightarrow AF\ door = open)$
05a	$EF(floor = 1 \wedge idle \wedge door = closed \wedge AX(door = closed))$
05b	$AG\ (floor = 1 \wedge idle \wedge door = closed \wedge AX\ (door = closed) \Rightarrow EG(floor = 1 \wedge door = closed))$
05-part	$EF(AX\ (door = closed))$
05c	$EF(floor = 3 \wedge idle \wedge door = closed \wedge AX\ (door = closed))$
05d	$AG\ (floor = 3 \wedge idle \wedge door = closed \wedge AX\ (door = closed) \Rightarrow EG(floor = 3 \wedge door = closed))$
05e	$EF(EG(door = closed))$
05′	$\neg AG\ (floor = 4 \wedge idle \Rightarrow E[idle U floor = 1])$
06	$\neg AG\ ((floor = 3 \wedge \neg liftBut3.pressed \wedge direction = up) \Rightarrow door = closed)$
07	$\neg AG\ ((floor = 3 \wedge \neg liftBut3.pressed \wedge direction = down) \Rightarrow door = closed)$

6. Evaluation

In [36], the authors propose the fSMV language along with a verification technique for CTL that is based on the enumerative algorithm, i.e., an exhaustive enumeration of the set of products (although they limit themselves to couples of features). Such a product-enumerative approach is exactly what we intend to avoid. While both approaches produce equivalent results, we argue that model checking a single model with variability (i.e., the model of the whole SPL) is in general more efficient. Experiments with the Haskell FTS library also suggest that [15]. Here, we test this hypothesis in the symbolic context through benchmarks that compare the runtime of the enumerative algorithm and the FTS algorithm.

6.1. Elevator system

For these experiments we used the elevator system from [36]. We extended the SMV models provided with the original paper in two ways. First, we made the number of floors (fixed at five in [36]) variable. Secondly, we added four more features to the system, giving a total of nine features. All features are independent, which means that there are 2^9 products. The elevator system is comprised of a number of platform buttons and a number of cabin buttons. There is a single button on each platform, which calls the elevator. The button press is modelled non-deterministically, and a pressed button remains pressed until the elevator has served the floor and its doors are opened. The elevator will always serve all requests in its current direction before it stops and changes direction. When serving a floor, the lift doors open and close again. There are nine features that modify the behaviour of the lift. Those marked with an asterisk were added by us.

Anti-prank.* Normally, a lift button will remain pushed until the corresponding floor is served. With this feature, the lift buttons have to be held pushed by a person.
Empty. If the lift is empty, then all requests made in the cabin will be cancelled.
Executive floor. One floor of the building has priority over the other floors and will be served first, both for cabin and platform requests.
Open when idle.* When idle, the lift opens its doors.
Overload. The lift will refuse to close its doors when it is overloaded.
Park. When idle, the lift returns to the first floor.
Quick close.* The lift door cannot be kept open by holding the platform button pushed.
Shuttle.* The lift will only change direction at the first and last floor.
Two-thirds full. When the lift is two-thirds full, it will serve cabin calls before platform calls.

To test the correctness of our approach we first reduced the example to the five features from [36] and managed to reproduce the feature interactions reported in the original paper. Subsequently, we made some minor modifications to the model to accommodate the additional features. The models contains over 300 lines of code; the actual number depends on the number of floors. The models are distributed with the toolset on the FTS website http://www.info.fundp.ac.be/fts.

The properties used for our benchmarks are those of the base system shown in Table 1 (mostly combined safety and liveness properties). The property numbers reported in the statistics refer to the numbers in Table 1, and can also be used to identify the properties in the NuSMV code. For instance, Property `01` specifies that “each time the button at floor 2 is pressed, the elevator must eventually be at floor 2 with its door opened”.

Table 2
Benchmark results for the elevator system with four floors.

Property	Value	Enumerative	Single	Speedup
01	*false*	17.84 s	0.14 s	127.43
01′	*true*	15.37 s	0.05 s	307.40
04	*false*	18.19 s	1.06 s	17.16
02	*false*	19.23 s	0.22 s	87.41
03a	*false*	20.48 s	1.84 s	11.13
03b	*false*	21.23 s	1.76 s	12.06
05a	*false*	20.09 s	3.23 s	6.22
05b	*true*	14.36 s	0.03 s	478.67
05-part	*true*	16.47 s	0.06 s	274.50
05c	*false*	19.94 s	1.86 s	10.72
05d	*true*	14.68 s	0.03 s	489.33
05e	*false*	18.3 s	1.06 s	17.26
05′	*false*	19.89 s	1.62 s	12.28
06	*true*	18.89 s	1.2 s	15.74
07	*true*	19.27 s	2.57 s	7.50

6.2. Experimental setup

The goal of our experiments is to evaluate the performance of our algorithms. To this end, we compare the FTS algorithm with the enumerative algorithm.

The experiments consist in using both algorithms to check the fifteen properties of the base system given in Table 1 against an elevator model with the number of floors ranging from 4 to 8. Each property was benchmarked in a separate run of the model checker. The benchmarks were run on an Ubuntu machine with an Intel Core2 Duo at 2.80 GHz with 4 Gb of RAM.

The reported benchmarks compare (for each property)

- the total runtime of 2^9 model checks that enumerate all products explicitly ('Enumerative');
- the runtime of a single NuSMV model check following our method ('Single').

The size of the NuSMV model of the product with all features ranges from 2^{17} states for four floors, to 2^{27} states for eight floors. These are the upper bounds for the size of the models analysed in the *enumerative* benchmarks. As explained earlier, our algorithm only needs one check, but requires an additional variable for each feature. Its models are thus much larger, from 2^{26} states to 2^{36}.

An important factor in BDD based model checking is the variable ordering. In order to avoid computing static variable orderings and still be efficient, NuSMV has the parameter `-dynamic`, which causes the BDD package to reorder the variables during verification in case the BDD size grows beyond a certain threshold. When using this method for the single model check, it works well on small to medium models (up to six floors). However, its limitations become more and more apparent as the size of the models grows. In the case of the single model check for eight floors (i.e., a model of size 2^{36}), NuSMV would spend more time reordering variables than actually verifying the property.

In consequence, we used dynamic reordering to extract efficient variable orderings for all models, and next used these as static orderings in all subsequent benchmarks. The model checks of the *single* approach were run with parameters `-df -i orderfile`. Option `-i` allows us to specify a static variable ordering to be used by the model checker. Those of the *enumerative* approach were run with `-df -dynamic`. It is important to note that the variable orderings computed for the *single* approach cannot be reused for the *enumerative* case. This is due to the fact that the *enumerative* approach produces 2^9 models with different sets of variables, which would require 2^9 variable orderings. However, due to the absence of the nine feature variables, the individual models of the *enumerative* cases are much smaller than the single model in the *single* case. Therefore, the dynamic variable ordering, while being the only option, should still be rather efficient for the *enumerative* case.

Note that we were not able to detect any patterns regarding the placement of feature variables in the variable orderings. Moreover, when comparing the variable orderings for the models with different numbers of floors, the positions of feature variables appear to change randomly.

6.3. Results

The results of the benchmarks for the model with four floors are given in Table 2. The results for the other models can be found in Appendix A. They show that our approach achieves order-of-magnitude speedups over the enumerative approach. These observations are reported for each property in Fig. 3, where we show how speedup evolves when the number of floors grows. Three clusters appear: four high outliers, with speedups greater than 250 and up to 1000; five low outliers with speedups below two or three and sometimes negative; and six stable properties with speedups around ten. A trend that we observed is that with an increasing number of floors, the outliers tend to become more extreme (the high speedups

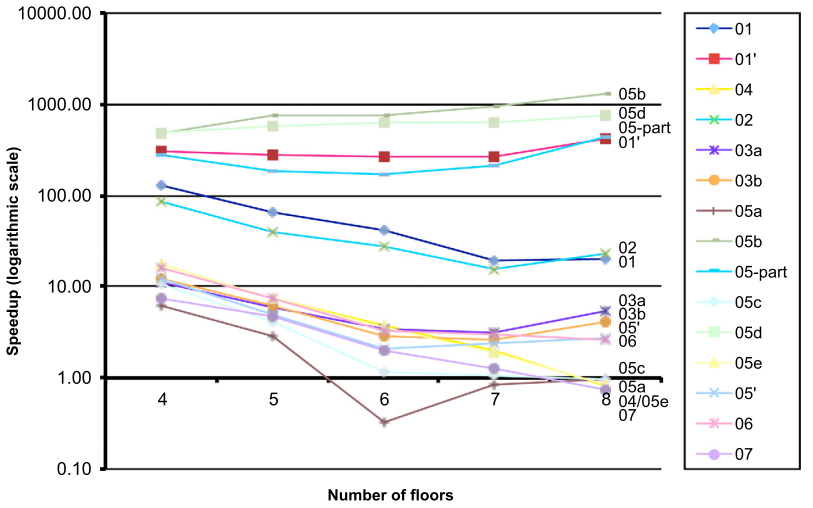

Fig. 3. Evolution of speedup with the number of floors (logarithmic scale).

grow, the low speedups descend). This is most likely due to the importance of the static variable ordering for larger models, although we cannot exclude other factors.

We conducted a second experiment in which we used the dynamic variable ordering for both algorithms. In this case, when the number of floors was larger than six, the smaller models of the enumerative algorithm caused it to be more efficient than the FTS algorithm on more than half of the properties. This means that the improvement in speedup seen above can be attributed to a large extent to the use of an optimised static variable ordering. The crucial advantage of our algorithm is therefore that it just needs one variable ordering. Note that both algorithms could be combined. First, our algorithm would be used to find a good variable ordering. Since this variable ordering comprises all the variables that are used in the enumerative checks, a variable ordering for each enumerative check can be obtained from this ordering by removing irrelevant variables. The enumerative algorithm could then be used with static orderings (obtained with the FTS algorithm). This procedure has not been tested nor subjected to benchmarks.

6.4. Threats to validity

In order to limit bias, we went to great lengths to ensure that the *enumerative* benchmarks were as efficient as possible. For instance, the computation of the 2^9 feature compositions (to create the files that were model checked) for each property was not included in the runtime. Furthermore, the large volume of log files from these runs was cleaned after each run since it would slow down the tool after several runs (because of huge inode lists in the parent folder).

NuSMV has an extensive set of optimisations and parameters of which we only used the most basic ones, `-dynamic` and `-df` (preventing the computation of reachable states, which was found to be slowing down both algorithms). The enumerative algorithm might benefit more from some of these optimisations, since even a small improvement can accumulate over the $O(2^n)$ runs of the tool. On the other hand, since the set of variables is different for most products, some optimisations cannot be used in the enumerative algorithm. For example, NuSMV allows hard caching of BDDs, so that they can be reused between checks. This caching cannot be used between products, because most products have different variables and thus different BDDs. Using the basic algorithm allowed us to make sure that the only difference in runtime was due to the use of the FTS algorithm and a static variable ordering.

7. Related work

7.1. FTSs

We first compare this paper with our previous work on FTSs. In [15], we introduced FTSs as a foundational formalism for behavioural specification and verification of SPLs. We focused exclusively on *linear* time logic properties as well as on *explicit-state* algorithms that exploit the compact structure of FTSs for efficient model checking. The paper [12] is an extension of [15] where the definition of FTSs is generalized and more information are covered. In particular, we studied the concept of parallel composition and provide details about the model checking algorithms. Yet these algorithms are based on an explicit visit of each state, whereas the present paper focuses on symbolic model checking. In [11] we exhaustively introduced SNIP, a tool that implements the theory presented in [12]. Again, the algorithms in SNIP are not symbolic, hence the motivation of the toolset proposed in this paper.

In [14], we lay the theoretical foundations for *fully symbolic* model checking of SPLs using NuSMV. The present paper extends the latter by studying a high-level modelling language, *viz.* fSMV, and its relation with FTSs. The expressiveness equivalence between the fSMV and FTSs is an important result that is missing in [14]. Additional contributions of the present paper are a thorough description of our toolset, and new empirical results. Overall, this paper substantially adds to the set of arguments in favour of the appropriateness (through the proof of equivalence), correctness (idem), computational efficiency (through the empirical evaluation) and scalability (through symbolic algorithms) of the set of tools and languages we propose for product line modelling and model checking.

In recent work, we define abstractions for FTSs through the definition of behavioural preorder between FTSs [19]. Based on that result, we lay the foundations for incremental verification of SPLs [18].

7.2. Behavioural models for SPLs

Apart from FTSs, there exist other formalisms to model SPLs behaviour. Larsen et al. [31] use I/O automata to model SPLs whose products are opened systems. They also define simulation/refinement relations between SPLs. However, they are not concerned with model-checking. Several authors propose modal transition systems [24,22,23,3] to model and verify the behaviour of SPLs. However, those models do not allow to keep track of the features (hence of the products) during a verification. Asirelli et al. [2] compare this approach with our work and highlight commonalities and differences between them.

7.3. High-level languages for SPLs specification

Several authors propose high-level languages for modelling SPL behaviour [41,20]. However, they do not consider model-checking. On the contrary, fSMV is the only high-level language for behavioural modelling of SPLs combined with efficient model checking algorithms. The main difference between the original work by Plath and Ryan [36] and ours is that we focus on modelling and verifying *multiple* products at once, while they are interested in feature interaction detection. More precisely, the approach in [36] consisted in checking feature combinations exhaustively, but limited to *pairs* of features. Our approach consists in using a reduction to FTSs in order to verify all the products with a single application of the model checking algorithm. Finally, in [37], Plath and Ryan already proposed a formal model for *fSMV*. However, we argue that the translation we proposed in Section 4 is compositional and easier to understand.

7.4. SPLs model-checking

Lauenroth et al. [32] introduce a CTL model-checking algorithm for verifying automata labelled with features. This algorithm is not symbolic and the formalism they use do not allow to label transitions with arbitrary boolean expressions. Gruler et al. [27] propose an extension of CCS with a variability operator that allows to model alternative choices. Their model-checking algorithm is, to the best of our knowledge, only sketched and not implemented. The approach of Apel et al. [1] allows to detect interactions between features, which are specified modularly. Like ours, their algorithms avoid redundant verifications. A comparable approach, which has been largely discussed in the previous section, is the compositional method developed by Fisler and Krishnamurthi [25,30], further extended by Li et al. [33]. Modular feature verification was not in the scope of our work until now, but comparing and combining our approach with modular verification is an exciting line of research for the future. Treating features in a specific way depending on their level of “cross-cuttingness” seems both a promising and challenging endeavour.

Other tools similar to our NuSMV extension exist. SNIP is our other SPL model checker [11]. It implements the semi-symbolic algorithms we presented in [15,12]. VMC [40] is a model checker that implements the MTS-based approach of Asirelli et al. [2].

8. Conclusion

In this paper, we tackled the state explosion problem in SPL model checking by introducing symbolic (as opposed to explicit) algorithms. Our experiments show that these algorithms achieve order-of-magnitude reduction in the verification time with regard to an enumerative verification of all the products. We also studied the relationship between FTSs, a formal model for SPL behaviour [15], and fSMV, a feature-oriented extension of the SMV language proposed in [36]. The main drawback of FTSs is that it is a foundational formalism, which is impractical for direct usage by engineers. The results of this paper establish that the fSMV language can be used as a high-level representation of FTSs. This work thus constitutes a significant progress towards developing specification and verification techniques that are suitable for SPL engineers in industrial settings. To summarize, the contributions of the paper are:

1. We defined the fCTL logic, which allows us to verify temporal properties on a specific set of products, and we designed symbolic algorithms for checking an FTS against an fCTL formula.
2. We gave a new compositional formal semantics to the fSMV language that is suited for SPL model checking.
3. We proved the expressiveness equivalence between fSMV and FTSs. The constructive proof is the basis for trustworthy translation algorithms.
4. We illustrated and evaluated some practical implications of the above results through our toolset and case study.

These contributions are part of a long term research project whose objective is to propose new languages and methods for the design and the efficient quality assessment of SPLs. As part of the project, we are also investigating linear temporal logic and explicit-state model checking through a feature-oriented extension of the Promela language (used in SPIN [28]).

In the future, we plan to apply compositional reasoning to the verification of SPL. Our goal is to identify special classes of features that can be verified *in isolation*, to propose algorithms that analyze the specification of such features and subsequently determine whether feature is responsible for the violation of a given property.

Appendix A. Detailed benchmark results

In this appendix, we provide details (see Tables A.3–A.6) about the results of the experiments discussed in Section 6. Basically, for each variant of the elevator model, we present a table comparing the performance of our symbolic algorithm with regard to an individual verification of each product of the SPL.

Table A.3
Benchmark results for the elevator system with five floors.

Property	Value	Enumerative	Single	Speedup
01	*false*	29.38 s	0.44 s	66.77
01′	*true*	24.76 s	0.09 s	275.11
04	*false*	34.02 s	4.62 s	7.36
02	*false*	33.16 s	0.82 s	40.44
03a	*false*	37.98 s	6.3 s	6.03
03b	*false*	39.43 s	6.32 s	6.24
05a	*false*	39.77 s	13.99 s	2.84
05b	*true*	22.7 s	0.03 s	756.67
05-part	*true*	29.25 s	0.16 s	182.81
05c	*false*	35.52 s	8.66 s	4.10
05d	*true*	23.44 s	0.04 s	586.00
05e	*false*	34.09 s	4.63 s	7.36
05′	*false*	40.21 s	8.14 s	4.94
06	*true*	34.55 s	4.56 s	7.58
07	*true*	35.9 s	7.57 s	4.74

Table A.4
Benchmark results for the elevator system with six floors.

Property	Value	Enumerative	Single	Speedup
01	*false*	44 s	1.05 s	41.90
01′	*true*	34.02 s	0.13 s	261.69
04	*false*	67.76 s	18.44 s	3.67
02	*false*	52.36 s	1.87 s	28.00
03a	*false*	76.67 s	22.42 s	3.42
03b	*false*	77.98 s	27.21 s	2.87
05a	*false*	105.07 s	322.53*s*	0.33
05b	*true*	30.67 s	0.04 s	766.75
05-part	*true*	54.63 s	0.32*s*	170.72
05c	*false*	88.63 s	78.36 s	1.13
05d	*true*	30.93 s	0.05 s	618.60
05e	*false*	67.45 s	18.39 s	3.67
05′	*false*	131.78 s	63.61*s*	2.07
06	*true*	68.36 s	20.42 s	3.35
07	*true*	73.06 s	36.89 s	1.98

Table A.5
Benchmark results for the elevator system with seven floors.

Property	Value	Enumerative	Single	Speedup
01	*false*	66.89 s	3.45 s	19.39
01′	*true*	44.34 s	0.17 s	260.82
04	*false*	214.75 s	109.67 s	1.96
02	*false*	86.98 s	5.58 s	15.59
03a	*false*	160.43 s	51.35 s	3.12
03b	*false*	169.91 s	66.45 s	2.56
05a	*false*	487.98 s	571.69 s	0.85
05b	*true*	38.39 s	0.04 s	959.75
05-part	*true*	114.38 s	0.55 s	207.96
05c	*false*	269.19 s	257.98 s	1.04
05d	*true*	38.62 s	0.06 s	643.67
05e	*false*	214.13 s	112.79 s	1.90
05′	*false*	568.56 s	241.53 s	2.35
06	*true*	142.42 s	48.37 s	2.94
07	*true*	160.3 s	128.84 s	1.24

Table A.6
Benchmark results for the elevator system with eight floors.

Property	Value	Enumerative	Single	Speedup
01	*false*	99.14 s	4.96 s	19.99
01′	*true*	62.71 s	0.15 s	418.07
04	*false*	337.47 s	414.32 s	0.81
02	*false*	139.58 s	6.06 s	23.03
03a	*false*	312.05 s	57.65 s	5.41
03b	*false*	332.49 s	81.35 s	4.09
05a	*false*	2180.58 s	2232.39*s*	0.98
05b	*true*	51.26 s	0.04 s	1281.50
05-part	*true*	211.63 s	0.48 s	440.90
05c	*false*	851.58 s	899.2 s	0.95
05d	*true*	52.27 s	0.07 s	746.71
05e	*false*	337.81 s	407.84 s	0.83
05′	*false*	2441.67 s	887.8 s	2.75
06	*true*	263.68 s	102.39 s	2.58
07	*true*	325.31 s	439.25 s	0.74

References

[1] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, Dirk Beyer, Detection of feature interactions using feature-aware verification, in: Proceedings of the IEEE International Conference on Automated Software Engineering (ASE), IEEE Computer Society, 2011.
[2] Patrizia Asirelli, Maurice H. Ter Beek, Alessandro Fantechi, Stefania Gnesi, Formal description of variability in product families, in: Proceedings of the 15th International Software Product Line Conference, SPLC'11, Springer-Verlag, 2011, pp. 130–139.
[3] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, Alessandro Fantechi, A deontic logical framework for modelling product families, in: VaMoS'10, 2010, pp. 37–44.
[4] Christel Baier, Joost-Pieter Katoen, Principles of Model Checking, MIT Press, 2007.
[5] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, NuSMV 2.2 Tutorial, ITC-irst.
[6] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NuSMV: a new symbolic model checker, Int. J. Softw. Tools Technol. Transf. 2 (4) (2000) 410–425.
[7] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, in: Lect. Notes Comput. Sci., vol. 131, Springer, 1981, pp. 52–71.
[8] Andreas Classen, CTL model checking for software product lines in NuSMV, Technical report P-CS-TR SPLMC-00000002, University of Namur, 2010, available online.
[9] Andreas Classen, Modelling with FTS: a collection of illustrative examples, Technical report P-CS-TR SPLMC-00000001, University of Namur, 2010, available online.
[10] Andreas Classen, Quentin Boucher, Patrick Heymans, A text-based approach to feature modelling: Syntax and semantics of TVL, in: Special Issue on Software Evolution, Adaptability and Variability, Sci. Comput. Program. 76 (12) (2011) 1130–1143.
[11] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Model checking software product lines with SNIP, Int. J. Softw. Tools Technol. Transf. 14 (5) (2012) 589–612.
[12] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, Jean-François Raskin, Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking, IEEE Trans. Softw. Eng. 39 (8) (August 2013) 1069–1089.
[13] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, What's in a feature: A requirements engineering perspective, in: FASE'08, Held as Part of ETAPS'08, in: Lect. Notes Comput. Sci., vol. 4961, Springer, 2008, pp. 16–30.
[14] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, Symbolic model checking of software product lines, in: ICSE 33, ACM, 2011, pp. 321–330.
[15] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, Jean-François Raskin, Model checking lots of systems: Efficient verification of temporal properties in software product lines, in: ICSE 32, ACM, 2010, pp. 335–344.
[16] Andreas Classen, Patrick Heymans, Thein Than Tun, Bashar Nuseibeh, Towards safer composition, in: ICSE 31, Companion Volume, IEEE, 2009, pp. 227–230.
[17] Paul C. Clements, Linda Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[18] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, Managing evolution in software product lines: A model-checking perspective, in: Proceedings of VaMoS'12, ACM, 2012, pp. 183–191.
[19] Maxime Cordy, Andreas Classen, Gilles Perrouin, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, Simulation-based abstractions for software product-line model checking, in: Proceedings of ICSE'12, IEEE, 2012, pp. 672–682.
[20] Krzysztof Czarnecki, Michal Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: GPCE'05, in: Lect. Notes Comput. Sci., vol. 3676, 2005, pp. 422–437.
[21] Christof Ebert, Capers Jones, Embedded software: Facts, figures, and future, Computer 42 (4) (2009) 42–52.
[22] Alessandro Fantechi, Stefania Gnesi, A behavioural model for product families, in: ESEC-FSE'07, Companion, ACM, 2007, pp. 521–524.
[23] Alessandro Fantechi, Stefania Gnesi, Formal modeling for product families engineering, in: SPLC 2008, IEEE CS, 2008, pp. 193–202.
[24] Dario Fischbein, Sebastian Uchitel, Victor Braberman, A foundation for behavioural conformance in software product line architectures, in: ROSATEA '06, ISSTA 2006 Workshop, ACM, 2006, pp. 39–48.
[25] Kathi Fisler, Shriram Krishnamurthi, Modular verification of collaboration-based software designs, in: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9, ACM, New York, NY, USA, 2001, pp. 152–163.
[26] Nissim Francez, Ira Forman, Superimposition for interacting processes, in: Concur'90, in: Lect. Notes Comput. Sci., vol. 458, 1990, pp. 230–245.
[27] Alexander Gruler, Martin Leucker, Kathrin Scheidemann, Modeling and model checking software product lines, in: IFIP WG 6.1 FMOODS '08, Springer, 2008, pp. 113–131.
[28] Gerard J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2004.
[29] Kyo Kang, Sholom Cohen, James Hess, William Novak, Spencer Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical report CMU/SEI-90-TR-21, SEI, 1990.
[30] Shriram Krishnamurthi, Kathi Fisler, Michael Greenberg, Verifying aspect advice modularly, in: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, 2004, pp. 137–146.
[31] Kim Guldstrand Larsen, Ulrik Nyman, Andrzej Wasowski, Modal I/O automata for interface and product line theories, in: Rocco De Nicola (Ed.), ESOP, in: Lect. Notes Comput. Sci., vol. 4421, Springer, 2007, pp. 64–79.
[32] Kim Lauenroth, Simon Töhning, Klaus Pohl, Model checking of domain artifacts in product line engineering, in: IEEE/ACM ASE '09, 2009, pp. 269–280.
[33] Harry C. Li, Shriram Krishnamurthi, Kathi Fisler, Verifying cross-cutting features as open systems, in: SIGSOFT FSE, 2002, pp. 89–98.
[34] Kenneth McMillan, Symbolic Model Checking, Kluwer, 1993.
[35] Christos H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
[36] Malte Plath, Mark Ryan, Feature integration using a feature construct, Sci. Comput. Program. 41 (1) (2001) 53–84.
[37] Malte Plath, Mark Dermot Ryan, The feature construct for SMV: Semantics, in: FIW VI, IOS Press, 2000, pp. 129–144.
[38] Hendrik Post, Carsten Sinz, Configuration lifting: Verification meets software configuration, in: ASE'08, IEEE CS, 2008.
[39] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, Yves Bontemps, Feature diagrams: a survey and a formal semantics, in: RE'06, IEEE CS, 2006, pp. 139–148.
[40] Maurice H. ter Beek, Franco Mazzanti, Aldi Sulova, VMC: A tool for product variability analysis, in: Proceedings of FM '12, in: Lect. Notes Comput. Sci., vol. 7436, 2012, pp. 450–454.
[41] Tewfik Ziadi, Loïc Hélouët, Jean-Marc Jézéquel, Towards a UML profile for software product lines, in: Int. Workshop on Product Family Engineering (PPE), in: Lect. Notes Comput. Sci., vol. 3014, 2003, pp. 129–139.